|  |  |  |
| --- | --- | --- |
| Csci63 2017 | May 12 2017  IOT Streaming project report | |
| IOT streaming platform for handling real-time flight data analytics using Kafka , Spark , Cassandra , Zeppelin | | Mohan Rayapuvari Ryan Tischer Ramesh Maddi |

# Project summary:

**Real-Time Airline Flight Data with Kafka and Streaming. Examples of IOT and Data Analytics**

**Summary**

The goal of project is to demonstrate real-time analytics of streaming data about airline flights.

**Project Provides**:

* Data can help passengers, airlines and surrounding cities deal with delayed flights.
* Airlines address unhappy customers and improve the services.
* Surrounding cities can plan for more demand (hotels, rental cars, restaurants , etc)
* Passengers are notified

**Project Workflow**:

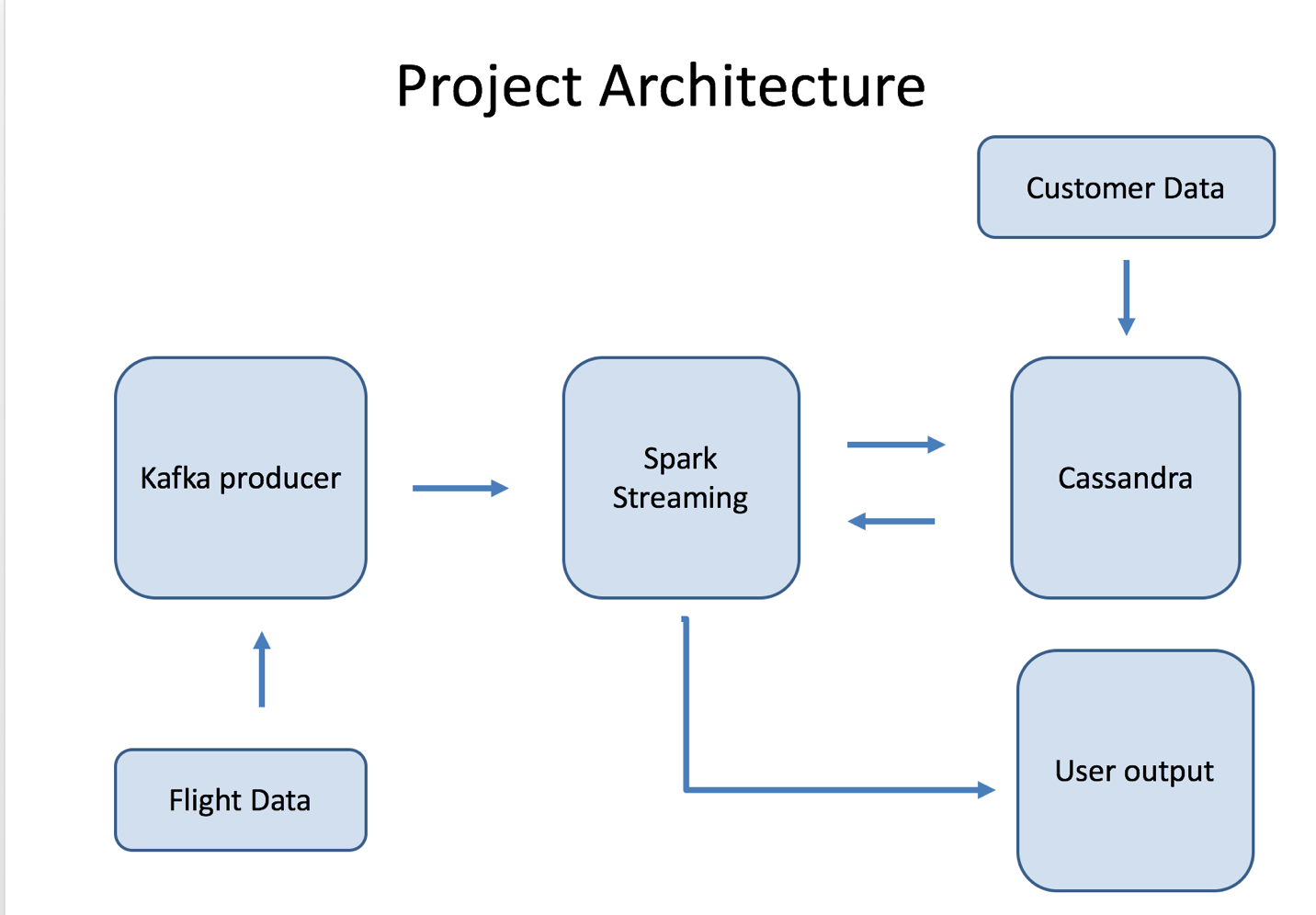
* Data downloaded from [https://www.transtats.bts.gov](https://www.transtats.bts.gov/)
* Data set is ~ 30 Columns and 2.5 million rows
* Data is broken up into 6 files
* Data is read in Spark Streaming as dsteam
* To simulate real-time data we have a Kafka producer send 500 rows every 1 seconds
* Dstreams are filtered to find flights that are delay
* Delayed flights are compared to list of customers (by flight number) in Cassandra to form new table
* Customer contact information and city of delay are displayed
* Zeppelin Dashboard to explore data

YouTube URL of the full presentation video:

YouTube URL of the 2min preview presentation video:

# Technology summary:

|  |  |  |
| --- | --- | --- |
| **Streaming** | **Kafka 0.8.2.1** | **AWS Cluster (1 zookeeper , 1 Broker)** |
| **Producer** | **Python 3.6; Kafka-python** | **Deployed to Broker** |
| **Big data platform** | **Spark 2.1** | **AWS EMR (1 Master , 2 Core servers)** |
| **Consumer** | **Pyspark kafka** | **AWS EMR step executed client mode** |
| **Data storage** | **Cassandra 3.9, cqlsh 5.0.1** | **AWS Cassandra server** |
| **Data Analysis** | **Apache zeppelin 0.7.1** | **AWS EMR** |
| **Local desktop** | **Windows 10** |  |
| **Github** | **Code and file repository** | **https://github.com/mohanrkrishna/csci63-SparkstreamingIOT.git** |

****

# Infrastructure set up overview:

* Cloud – AWS free account
* Kafka cluster

AWS Cloud formation used to create Kafka cluster with 1 zookeeper server and 1 Broker server; Cloud formation has a template to provision these servers pre-configured with all settings required for kafka streaming like adding zookeeper server in server.properties of kafka broker

SSH into zookeeper server to create 1 topic “airline1” for this project

More details on set up at [details section](#_Infrastructure_set_up)

* Spark cluster

AWS emr-5.5.0 with **Hadoop distribution:**Amazon 2.7.3 ; **Applications:**Hive 2.1.1, Hue 3.12.0, Spark 2.1.0, Sqoop 1.4.6, Zeppelin 0.7.1

1 Master node (Driver running) ; 2 Cores (Workers with executors)

**Master:**Running1m4.xlarge ; **Core:**Running2c4.large

More details on set up at [details section](#_Infrastructure_set_up_1)

* Cassandra cluster

Bitnami image from AWS marketplace which provides an out-of-the-box Cassandra cluster

More details on set up at [details section](#_Infrastructure_set_up_2)

* Zeppelin

AWS emr has option to host zeppelin on master node. Zeppelin comes with spark interpreter which makes easy to run spark jobs from UI and use spark SQL for data analysis and visualizations. Configured to connect to Cassandra cluster to fetch data and analyze

More details on set up at [details section](#_Infrastructure_set_up_3)

# Design-Code overview:

**Kafka producer in python**

Function:

* Connect to Kafka broker and topic ‘airline1’
* Read flight stream data (for this project used transtats data in csv/zip file)
* Using pandas read data in chunks (500 at a time) instead of loading entire file to dataframe, avoids memory issues
* Clean data : Replace NA with zeros , change column names to lower case
* Transform : Convert to json format. This makes easier to handle schema in spark and load to cassandra using

Dataframes

* Send data : Send using kafka-python api function , sleep for 1 second and iterate through dataframe iterator

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | | |  | |  | |  |  | | | **from** kafka **import** KafkaProducer **import** time **import** zipfile **import** pandas **as** pd  producer = KafkaProducer(bootstrap\_servers=**'ec2-52-91-250-49.compute-1.amazonaws.com:9092'**) topic = **'airline1'** zfile = zipfile.ZipFile(**"On\_Time\_On\_Time\_Performance\_2017\_1.zip"**) filename = zfile.filelist[0].filename print(filename) filecontent = zfile.extract(filename) columns\_list=[**'Year'**, **'Quarter'**, **'Month'** ,**'DayofMonth'** ,**'DayOfWeek'**, **'FlightDate'**,  **'UniqueCarrier'** ,**'AirlineID'**, **'Carrier'** ,**'TailNum'** ,**'FlightNum'**,  **'OriginAirportID'**, **'OriginAirportSeqID'**, **'OriginCityMarketID'**, **'Origin'**,  **'OriginCityName'**, **'OriginState'**, **'OriginStateFips'**,**'OriginStateName'**,  **'OriginWac'**, **'DestAirportID'**, **'DestAirportSeqID'**, **'DestCityMarketID'** ,**'Dest'**,  **'DestCityName'**, **'DestState'**, **'DestStateFips'** ,**'DestStateName'**, **'DestWac'**,  **'CRSDepTime'**, **'DepTime'** ,**'DepDelay'** ,**'DepDelayMinutes'** ,**'DepDel15'**,  **'DepartureDelayGroups'** ,**'DepTimeBlk'**, **'TaxiOut'**, **'WheelsOff'**, **'WheelsOn'**,  **'TaxiIn'** ,**'CRSArrTime'** ,**'ArrTime'**, **'ArrDelay'**, **'ArrDelayMinutes'** ,**'ArrDel15'**,  **'ArrivalDelayGroups'** ,**'ArrTimeBlk'** ,**'Cancelled'**, **'CancellationCode'**,  **'Diverted'** ,**'CRSElapsedTime'**, **'ActualElapsedTime'**, **'AirTime'**,**'Flights'**,  **'Distance'** ,**'DistanceGroup'**, **'CarrierDelay'**, **'WeatherDelay'**, **'NASDelay'**,  **'SecurityDelay'**, **'LateAircraftDelay'**, **'FirstDepTime'** ,**'TotalAddGTime'**,  **'LongestAddGTime'**]  df\_itr=pd.read\_csv(filecontent,dtype={**'cancellationcode'**: str, **'div2airport'**: str,**'div2tailnum'**:str},usecols=columns\_list,iterator=**True**,chunksize=500)  **for** chunk **in** df\_itr:  df = pd.DataFrame(data=chunk, index=**None**)  df1=df.fillna(0)  df1.columns = map(str.lower, df1.columns)  df1\_bytes = df1.to\_json(orient=**'records'**)  producer.send(topic, df1\_bytes)  time.sleep(1) |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |  |  | | |

Key notes:

* Kafka topic has size limit of 1MB in (bytes set up in message.max.bytes = 1MB at broker); json with 1000 at a time crossed 1MB so streaming did not work. So had to reduce to 500 at a time
* log.retention.hours = 168 is default set in zookeeper setting to store history data in kafka. For testing purpose and save space , made it 5 minutes using log.retention.minutes=5
* Kafka producer executed from AWS broker server itself to avoid connectivity issues. Could be run remotely from laptop too provided AWS broker can be reached

**Spark consumer(Pyspark)**

Core Function:

* Using KafkaUtils , connect to Kafka broker and topic ‘airline1’
* Set up streamingcontext with 2 seconds batch duration and streaming window of length 2 seconds and sliding

interval 2 seconds

* *Customer table in cassandra holds details of customer leveraging notification services from third party company or*

*flight carrier itself; Requirement is to find impacted customers with flight delays real-time and further notify*

*dependent service provides like cabs, hotels, restaurants etc. And also store entire stream data to a repository for*

*further machine learning and predictions*

* Using spark-cassandra-connector from Datastax , read customers table into dataframe and cache it
* Process streaming data calling ‘processdstream’ function on flightwindow dstream; returns ‘customers\_delay’

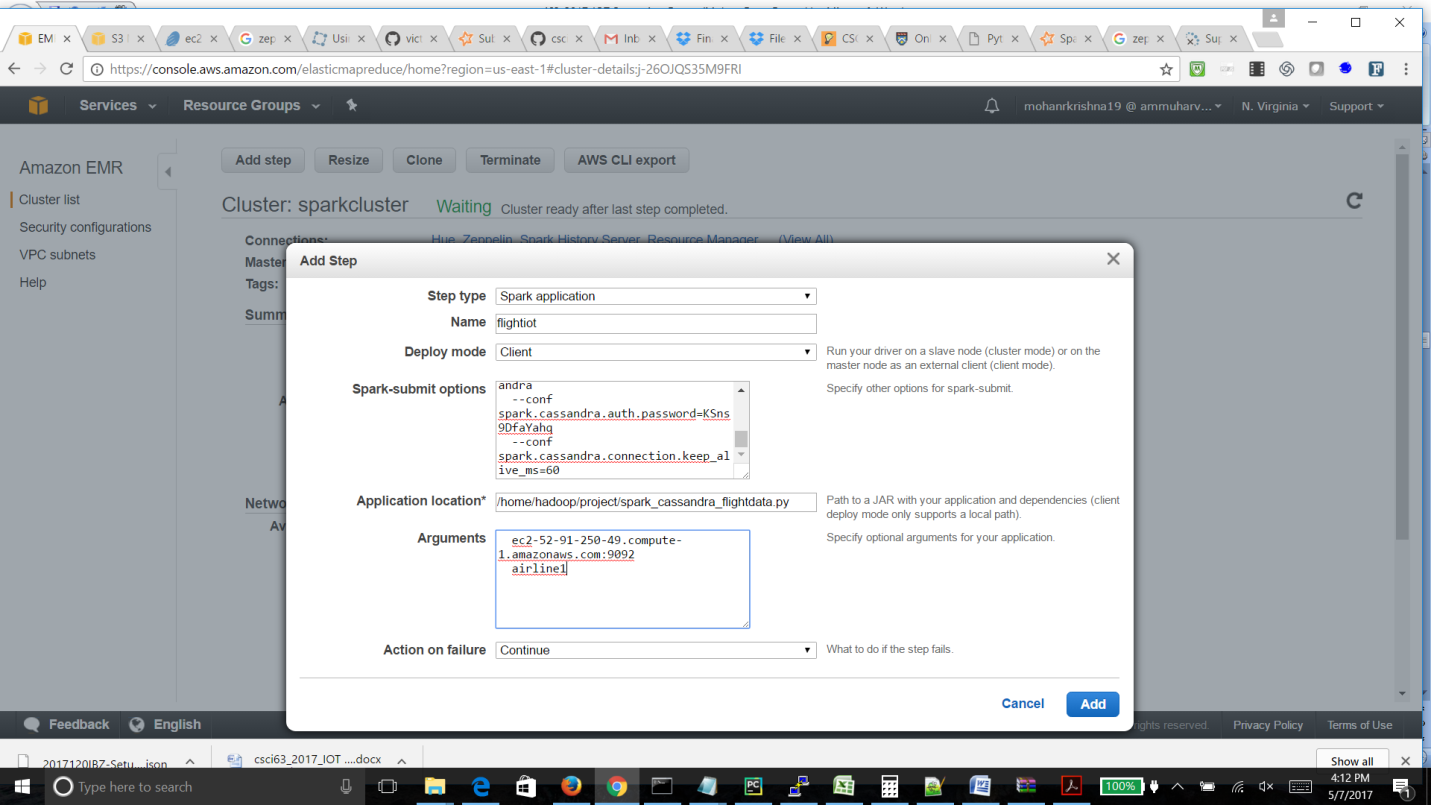
dstream joining flightwindow data (pick only delays) and customer data; functionality explained later

* Store flightwindow dstream to cassandra table calling ‘loadcassandra’ function
* Store customers\_delay dstream to cassandra table calling ‘loadcassandra’ function

|  |
| --- |
| **from** \_\_future\_\_ **import** print\_function **from** pyspark **import** SparkContext, SparkConf **from** pyspark.sql **import** SQLContext, Row **from** pyspark.streaming **import** StreamingContext **from** pyspark.streaming.kafka **import** KafkaUtils **import** json **import** sys  conf=SparkConf().setAppName(**"FlightIOTStreaming"**) sc = SparkContext(appName=**"FlightIOTStreaming"**) sql = SQLContext(sc) ssc = StreamingContext(sc, 2)  brokers, topic = sys.argv[1:]  flightstream = KafkaUtils.createDirectStream(ssc, [topic], {**"metadata.broker.list"**: brokers}) *#flightstream.pprint()* flightdata = flightstream.map(**lambda** kv:kv[1])  flightwindow = flightdata.window(2,2)  customer = sql.read.format(**"org.apache.spark.sql.cassandra"**). \  load(keyspace=**"flightkeyspace"**, table=**"customers"**).cache()  customers\_delay = flightwindow.transform(processdstream) print(customers\_delay.pprint()) flightwindow.foreachRDD(**lambda** time,rdd: loadcassandra(time,rdd,**"flightiot"**)) customers\_delay.foreachRDD(**lambda** time,rdd: loadcassandra(time,rdd,**"flightdelays"**))  sc.setCheckpointDir(**"hdfs:///user/spark/projects/iotcheckpoint/"**) ssc.start()  ssc.awaitTermination() |

Key notes:

* Amazon EMR provides ability to add a step from AWS website for running spark-submit; This avoids SSH into spark master or any other server to submit spark jobs
* Screenshot below: Choose step type – Spark application and give a name – ‘flightiot’; Deploymode : Choose client if driver program to be executed from master or cluster to run on worker nodes. Choose client for this run



* Spark-submit options (Copy below)

**--master yarn**

**--packages com.datastax.spark:spark-cassandra-connector\_2.11:2.0.1**

**--jars /home/hadoop/project/spark-streaming-kafka-0-8-assembly\_2.11-2.1.0.jar**

**--conf spark.cassandra.connection.host=ec2-54-226-206-176.compute-1.amazonaws.com**

**--conf spark.cassandra.auth.username=cassandra**

**--conf spark.cassandra.auth.password=KSns9DfaYahq**

**--conf spark.cassandra.connection.keep\_alive\_ms=60**

Using yarn as resource manager; Other options mesos or standalone spark manager

Spark-cassandra-connector from maven repository to be downloaded as package for cassandra connectivity

Download spark-streaming-kafka bin and place it in a folder at master

Other conf options for cassandra server , user id and password

* Application location = /home/hadoop/project/spark\_cassandra\_flightdata.py

Copy Pyspark code to above location

* Arguments (Pass Kafka broker and topic name)

ec2-52-91-250-49.compute-1.amazonaws.com:9092

airline1

* Clicking Add would start running spark streaming consumer in spark cluster. Run this step first before invoking kafka producer program
* In EMR cluster step section , links to spark-submit job provided to monitor status of the job. Spark server web ui can also be used
* For 0.5M records total process takes around 15-20 mins using our project clusters in AWS

Process dstream Function(Transformation):

* Invoked foreach rdd in flightdata sent by kafka; It is in json format
* Use sql.read.json to load into spark dataframe
* Extract only flight records with delays into another dataframe using filter function
* Join rdd of dstream with customer dataframe on columns flightnum , originalairportid , destairportid to identify

impacted customers with flight delay information (use .select to choose required columns)

* Results from join are stored in a dataframe. Need to convert it back to rdd as transform function expects rdd back to

create new dstream

|  |
| --- |
| **def processdstream(rdd):**  **if** (**not** rdd.isEmpty()):  flightdelays\_collist = [**'origin'**,**'flightnum'**,**'flightdate'**,**'email'**,**'origincityname'**,**'dest'**,**'destcityname'**,**'airtime'**,**'arrdelay'**,**'arrdelayminutes'**,**'arrivaldelaygroups'**,  **'arrtime'**,**'arrtimeblk'**,**'cancellationcode'**,**'cancelled'**,**'carrierdelay'**,**'departuredelaygroups'**,**'depdelay'**,**'depdelayminutes'**,**'deptime'**,**'deptimeblk'**,**'destairportid'**,**'deststate'**,  **'distance'**,**'distancegroup'**,**'diverted'**,**'first\_name'**,**'flights'**,**'id'**,**'last\_name'**,**'lateaircraftdelay'**,**'nasdelay'**,**'originairportid'**,**'originstate'**,**'phone'**,**'securitydelay'**,**'weatherdelay'**]  print(str(**'Processing flightdata...'**))   fstatusstream=sql.read.json(rdd)  fstatusstream\_delay=fstatusstream.filter((fstatusstream.depdelay !=0) | (fstatusstream.arrdelay !=0))  customers\_delay\_df = fstatusstream\_delay.join(customer,[**'flightnum'**,**'originairportid'**,**'destairportid'**],**'inner'**). \  select(flightdelays\_collist)  **return** customers\_delay\_df.rdd |

Loadcassandra Action:

* Used to load data to cassandra tables – Data = flightdata sent by kafka and customers\_delay data from above function
* Using cassandra connector , dataframe can be saved to cassandra table; columns should match
* Flightdata is already in json , so read json to df1 and load to cassandra table ‘flightiot’ using df1.write , keyspace and

table name provided in options ; save mode = append to add records , duplicates will be controlled using primary key

in cassandra. Other options are overwrite , ignore , error

* Customers\_delay rdd is not json , so convert to dataframe using toDF() and using same method to save to cassandra

table – ‘flightdelays’

impacted customers with flight delay information (use .select to choose required columns)

|  |
| --- |
| **def loadcassandra(time, rdd,ctable):**  *"""  Invoked by foreach against dtream to loop through all rdds;  """* **if** (**not** rdd.isEmpty()): *#Check if rdd not empty to avoid empty files* timesuffix = int(time.strftime(**'%s'**))  **if** ctable == **'flightdelays'**:  *#print(rdd.take(5))* df1 = rdd.toDF()  **elif** ctable == **'flightiot'**:  df1 = sql.read.json(rdd)   df1.write.format(**"org.apache.spark.sql.cassandra"**). \  options(table=ctable, keyspace=**"flightkeyspace"**). \  save(mode=**"append"**)   print(str(time.strftime(**'%x %X'**)) + **'...loading cassandra '**+ctable |

# Execution demo:

# Infrastructure set up details(Kafka):

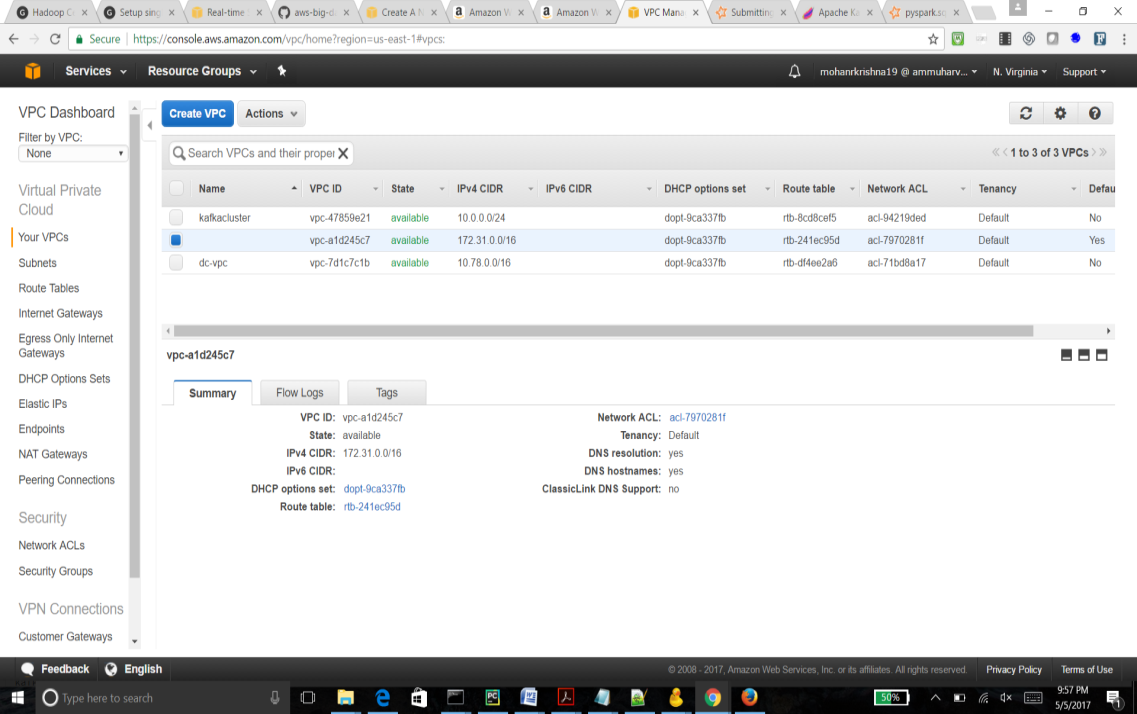
**Kafka cluster:**

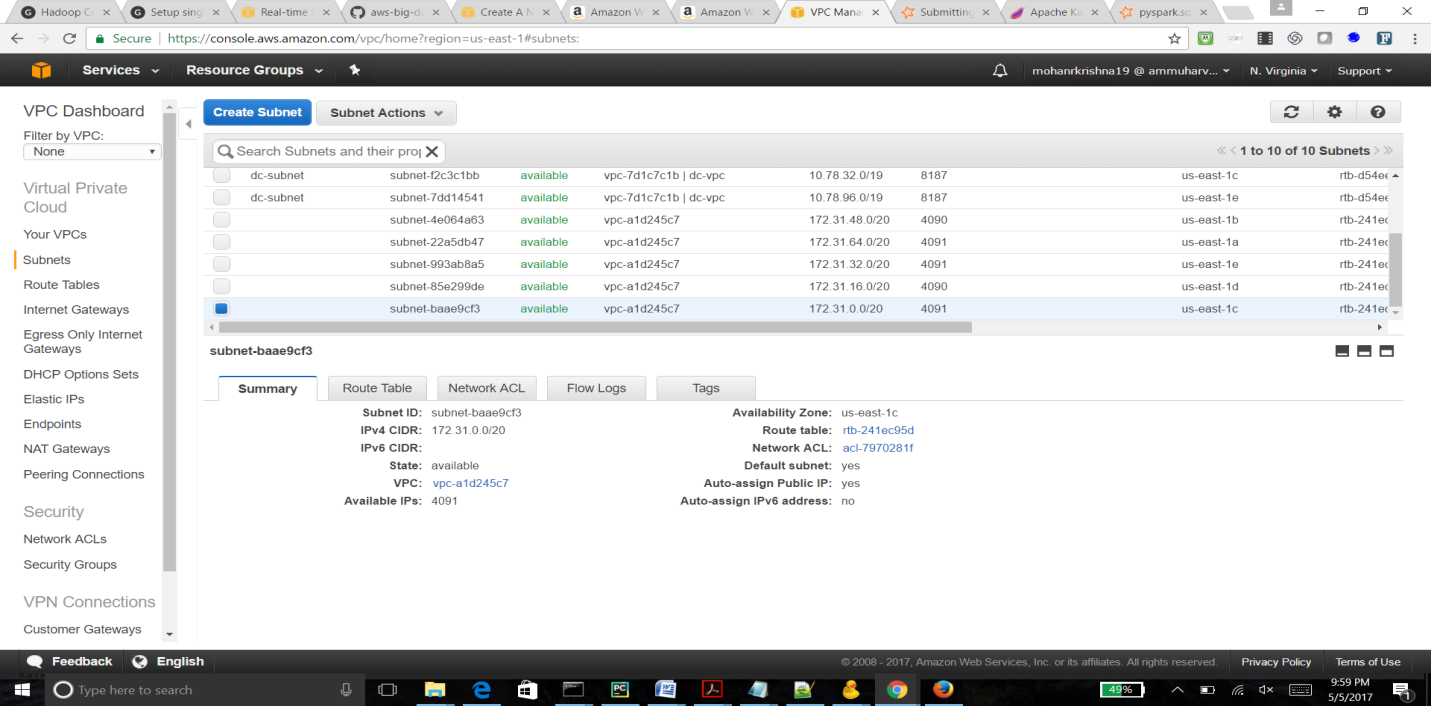
* Create following from AWS VPCto create Virtual private cloud and subnet for creating Kafka cluster
  + Create EC2 keyvalue pair for ssh; Download .pem file to local desktop
  + Create VPC and Subnet ; Important settings –

(VPC) DNS resolution: yes ; DNS hostnames:yes

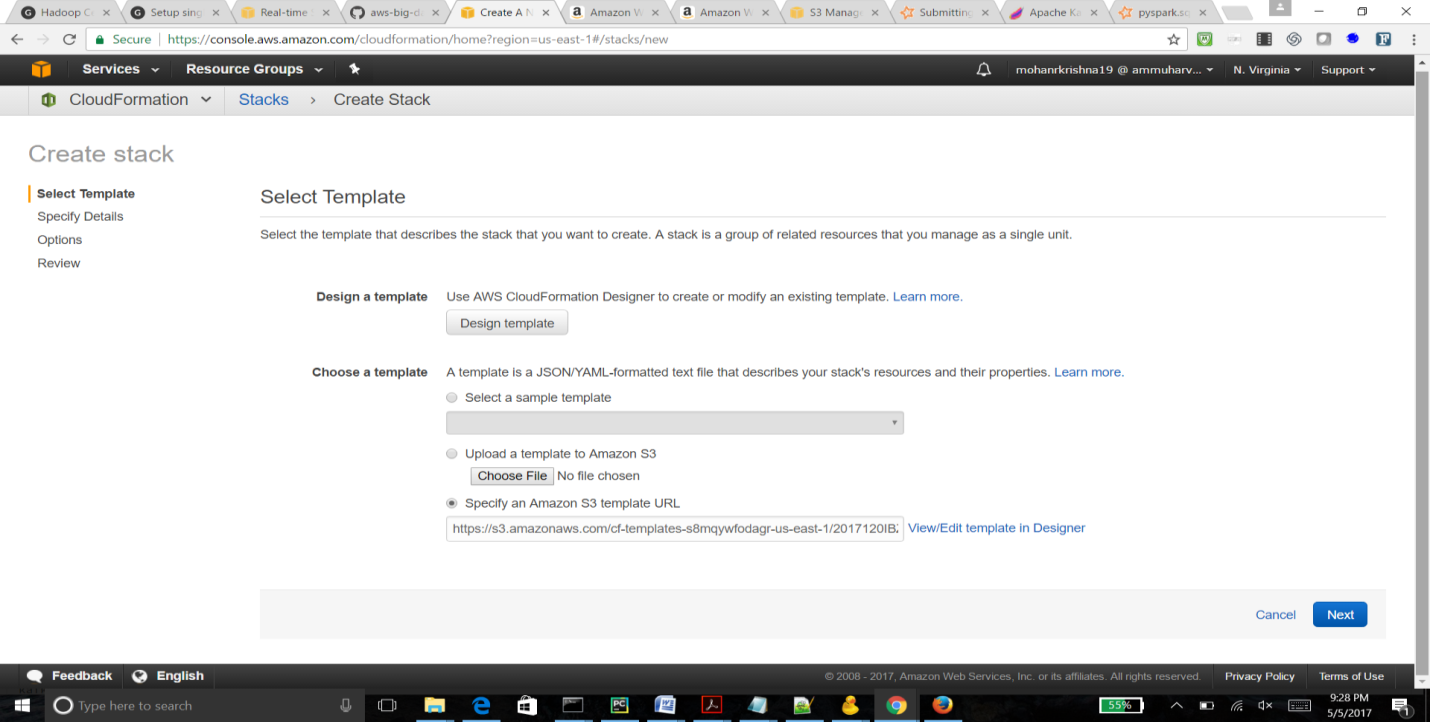
(Subnet) Auto-assign Public IP: Yes

|  |  |
| --- | --- |
|  |  |

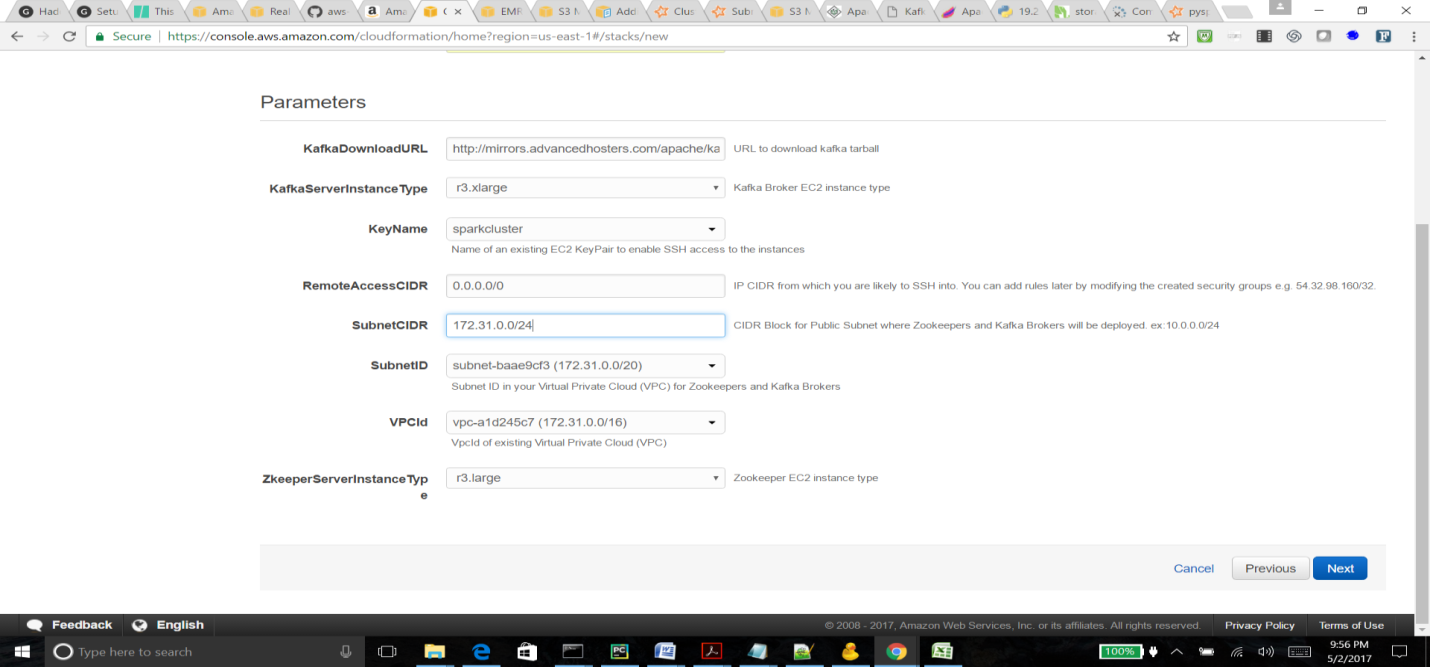




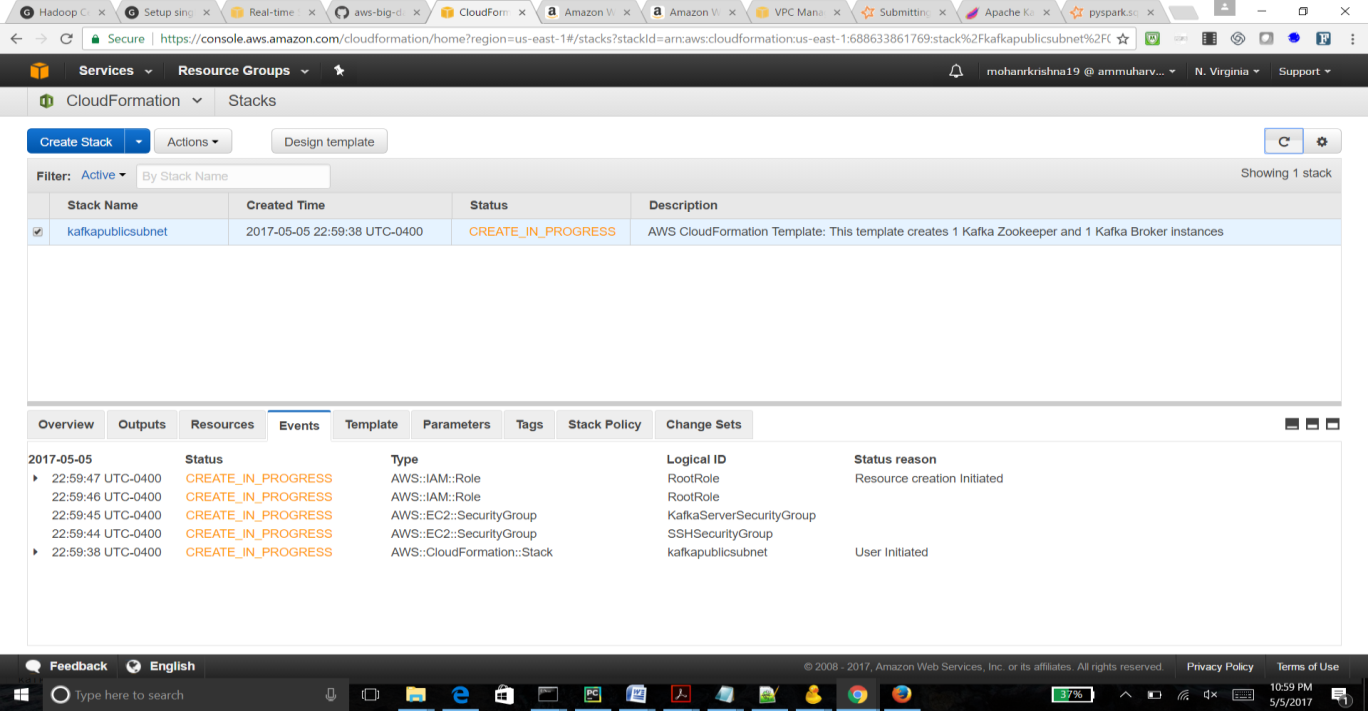
* Use json template at [CloudFormation template for public subnets](https://github.com/awslabs/aws-big-data-blog/blob/master/aws-blog-sparkstreaming-from-kafka/cloudformation/SetupKafka-AWS-PublicSubnets.json) ; Download to s3
* From AWS Cloud formation , create stack , choose above s3 json



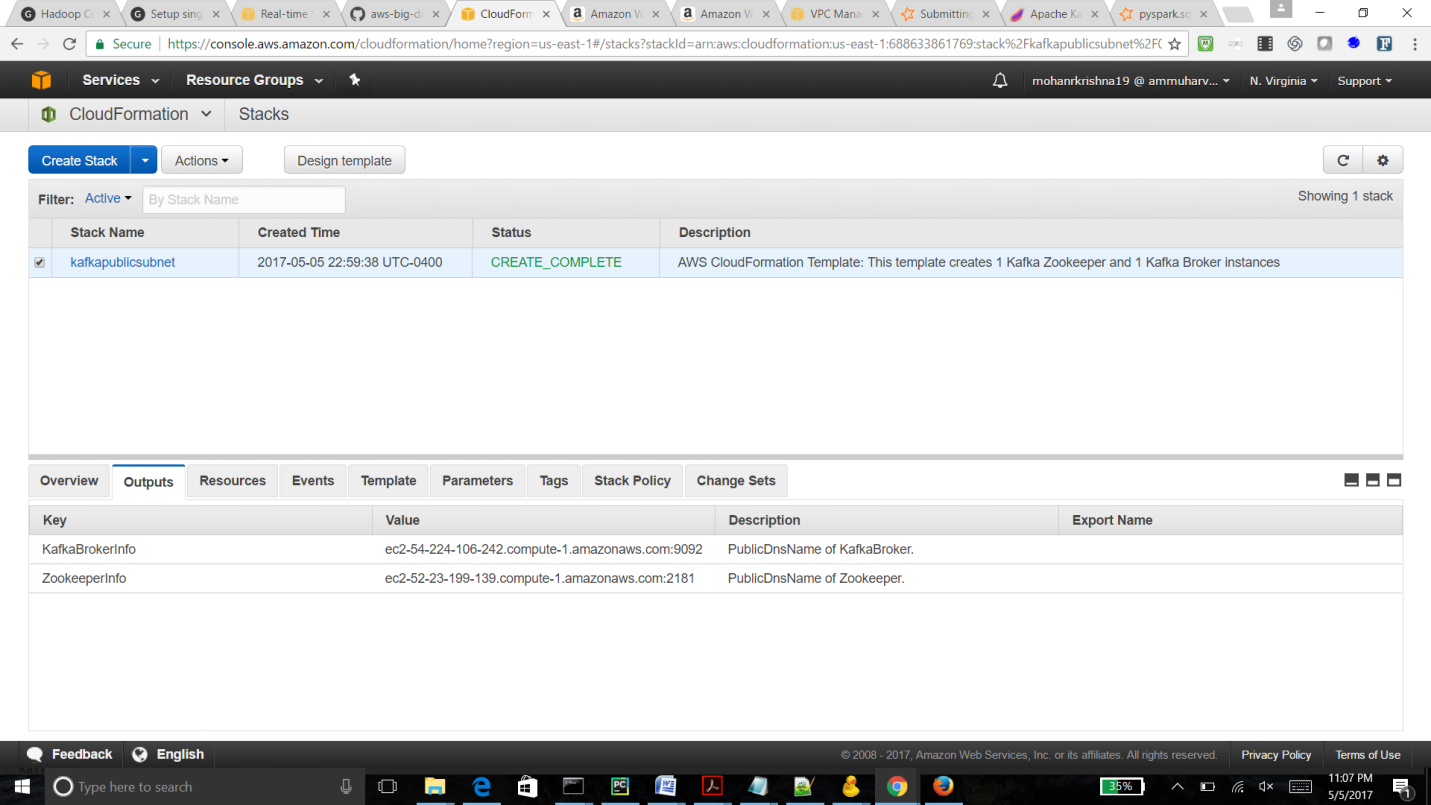
* Give “stackname”kafkapublicsubnet ;Select VPCid and subnet created above; Choose KeyName=EC2 keyvalue pair created. Important :SubnetCIDR should match Subnet CIDR , otherwise servers cannot be connected ; Keep Remote CIDR to 0.0.0.0/0 ; this is for laptops from which SSH is invoked. SourceIP can be added later too from EC2 menu



* Below screen shows progress , wait until ‘CREATE\_COMPLETE’



* Navigate to Outputs and save publicdns of KafkaBroker , Zookeeper servers , Required to login using SSH



**Kafka cluster create topic:**

* Install Git bash desktop app on windows
* Using Git bash , cd to directory with EC2 private key and remote login using SSH

email@LAPTOP-9CHTJ9IO MINGW64 /D/BigdataVMWare/Aws Keys

**$ ssh -i "sparkcluster.pem" ec2-user@ec2-52-23-199-139.compute-1.amazonaws.com**

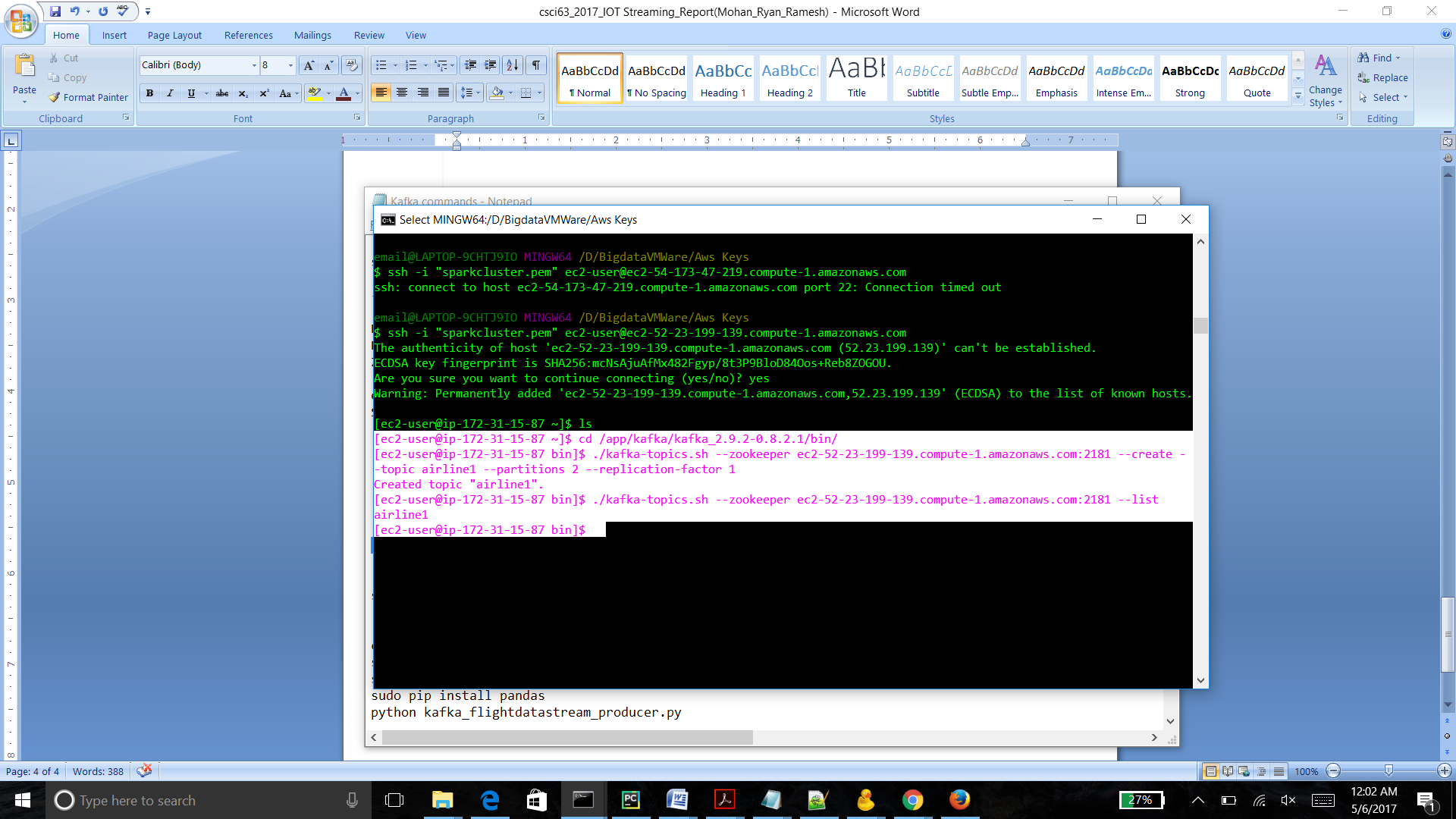
The authenticity of host 'ec2-52-23-199-139.compute-1.amazonaws.com (52.23.199.139)' can't be established.

ECDSA key fingerprint is SHA256:mcNsAjuAfMx482Fgyp/8t3P9BloD84Oos+Reb8ZOGOU.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'ec2-52-23-199-139.compute-1.amazonaws.com,52.23.199.139' (ECDSA) to the list of known hosts

* Navigate to kafka bin directory and create topic ; Verify topic created using --list

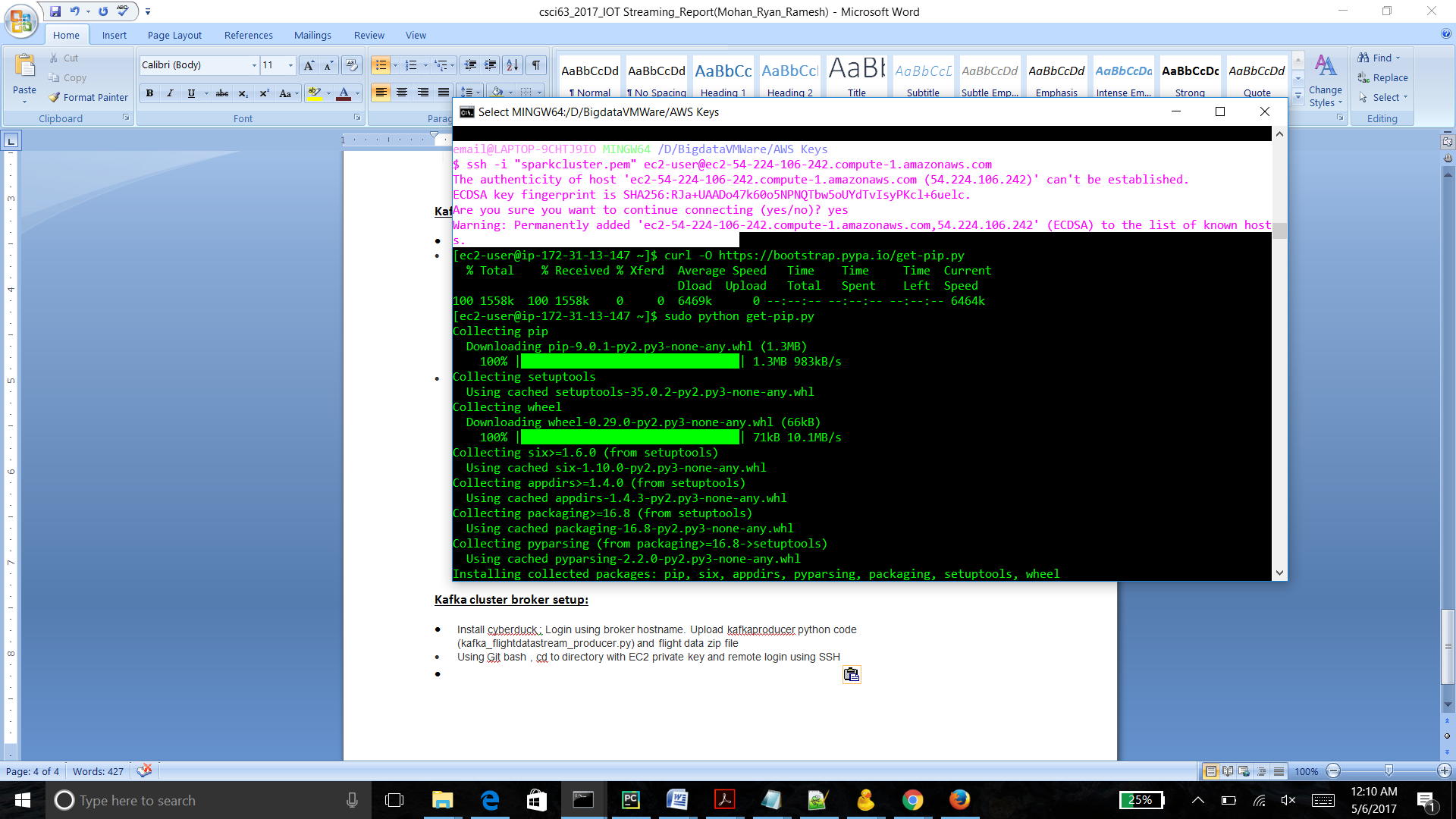


**Kafka cluster broker setup:**

* Install cyberduck ; Login using broker hostname. Upload kafkaproducer python code (kafka\_flightdatastream\_producer.py) and flight data zip file
* Using Git bash , cd to directory with EC2 private key and remote login using SSH

**email@LAPTOP-9CHTJ9IO MINGW64 /D/BigdataVMWare/AWS Keys**

**$ ssh -i "sparkcluster.pem"** [**ec2-user@ec2-54-224-106-242.compute-1.amazonaws.com**](mailto:ec2-user@ec2-54-224-106-242.compute-1.amazonaws.com)



* Kafka broker is used to run producer python program. So below python packages are required. Pip needs to be installed first followed by kafka-python , pandas

curl -O https://bootstrap.pypa.io/get-pip.py

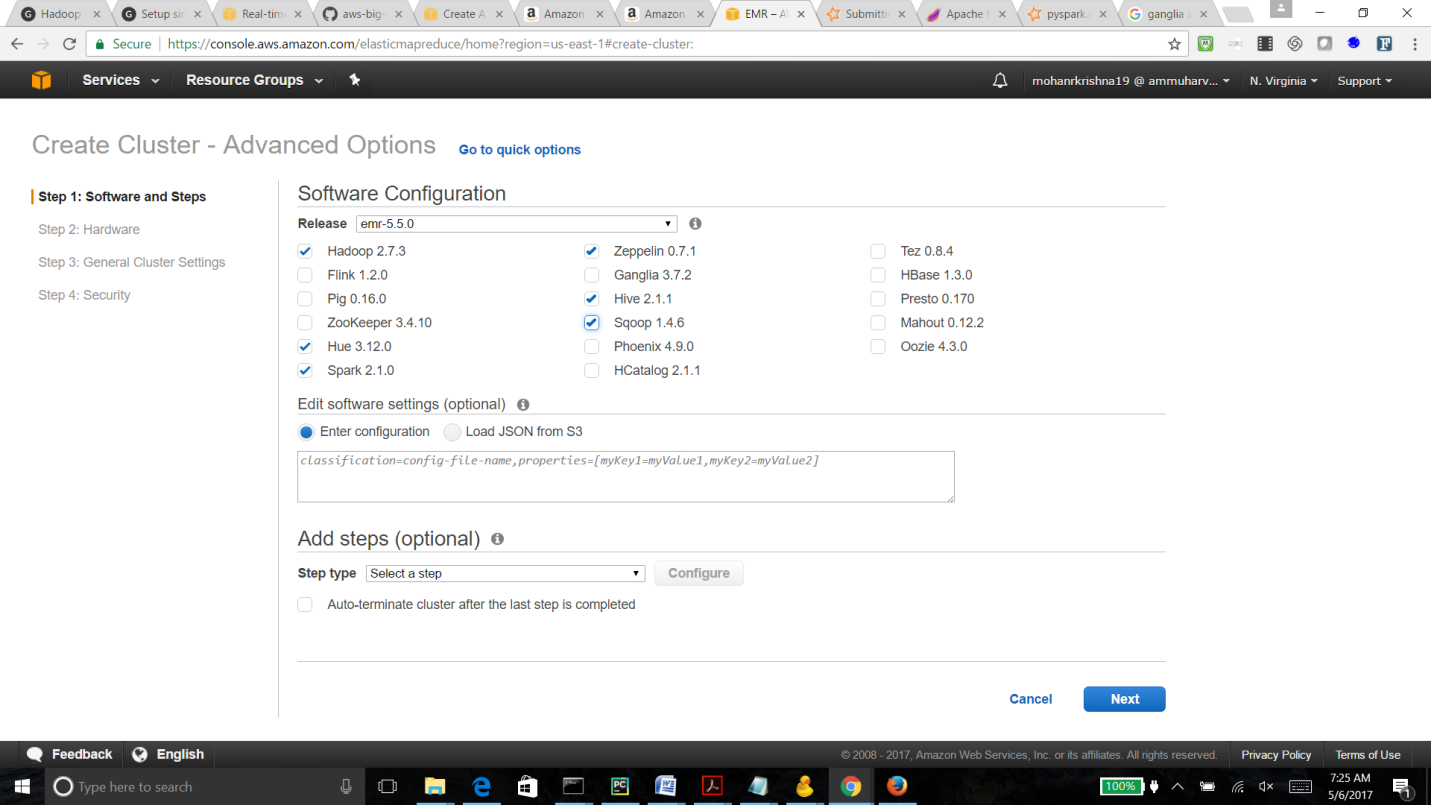
sudo python get-pip.py

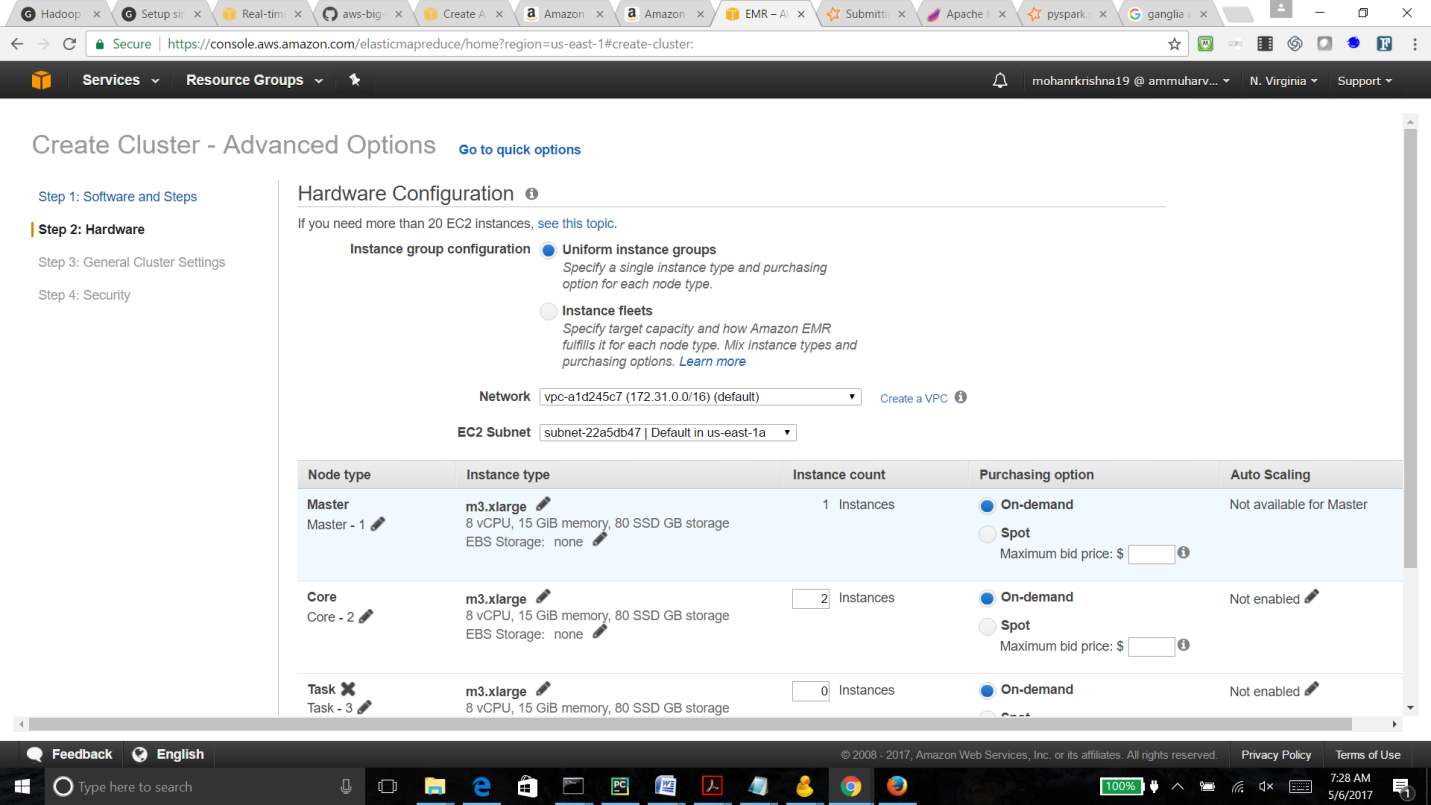
sudo pip install kafka-python

sudo pip install pandas

# Infrastructure set up details(Spark):

* Navigate to Amazon EMR (AWS)
* Click Create cluster ; give name of cluster – ‘sparkcluster’ and go to advanced options
* Choose latest emr – emr-5.5.0 , hadoop is default selected and required for spark to use HDFS , Yarn ; Choose Spark 2.1.0



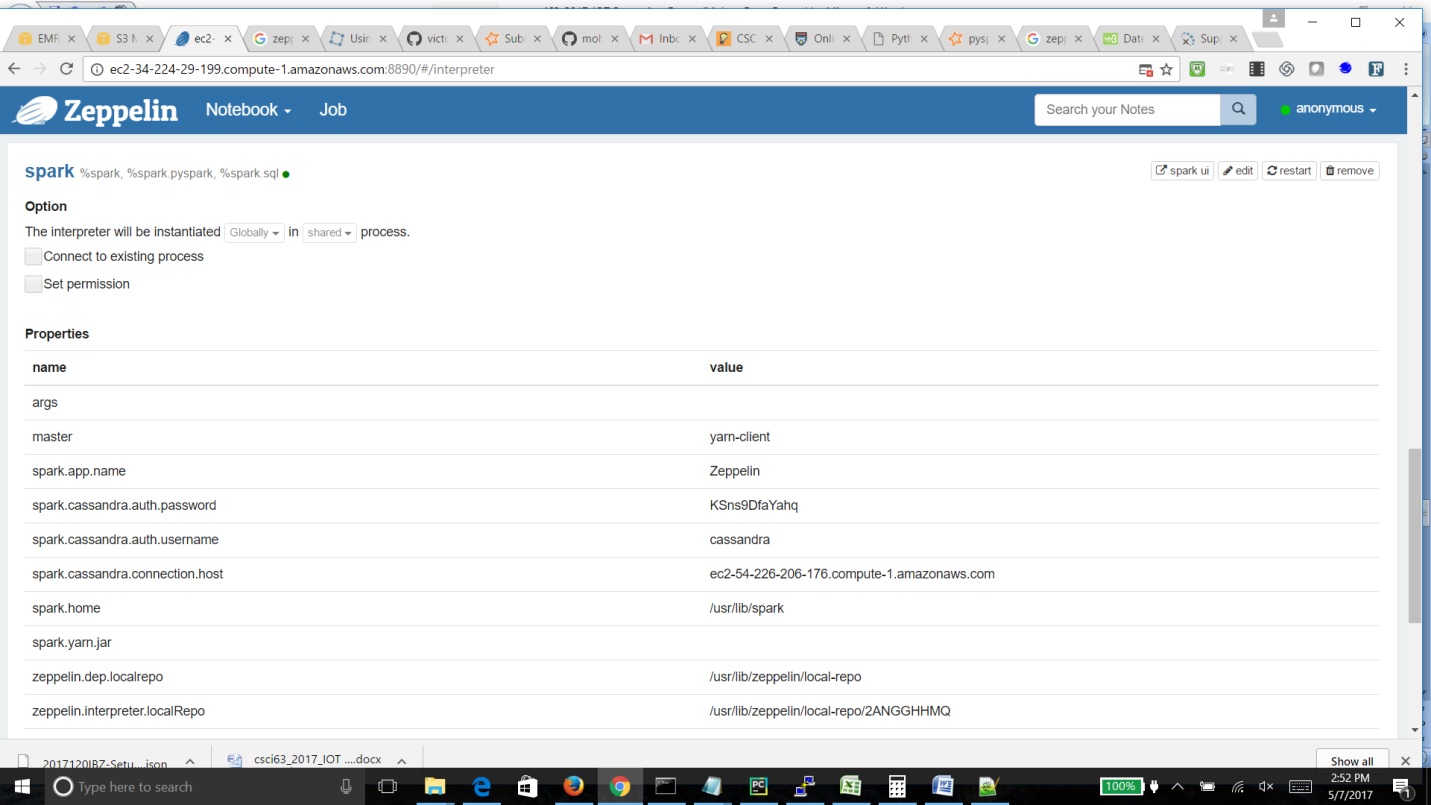
* Next hardware configuration , as this is small scale and pilot choose low scale servers to avoid charges ; Master node – m3.xlarge(8core , 15GB) , 2 Cores (for Spark Worker nodes, 2 Core , 3.8GB) ; Choose Network VPC and Subnet same as Kafka cluster to be able to avoid cross network access issues between spark and kafka servers
* Next general options , leave as-is
* Next Security Options , choose EC2 key value pair. I chose the same one used for Kafkacluster; Suggested to use different one
* Click Create cluster; It would take 15-20 mins

# Infrastructure set up details(Zeppelin):

AWS EMR has Web url to invoke zeppelin. Follow instructions to enable web connections to EMR cluster to access Zeppelin URL from laptop. In our case url is http://ec2-34-224-29-199.compute-1.amazonaws.com:8890

# 

Zeppelin has interpreters to integrate with various platforms. Spark is default. Interpreters can be accessed from webui. For this project , this tool is used for data analysis fetching from cassandra tables & utilizing spark for distributed computing. So needs to configure interpreter to connect to cassandra. And spark connection comes pre-configured as it is part of EMR



Add below three entries as properties as in above screen

--conf spark.cassandra.connection.host=ec2-54-226-206-176.compute-1.amazonaws.com

--conf spark.cassandra.auth.username=cassandra

--conf spark.cassandra.auth.password=KSns9DfaYahq

PySpark needs spark-cassandra-connector which provides api for cassandra based operations leveraging dataframes in spark

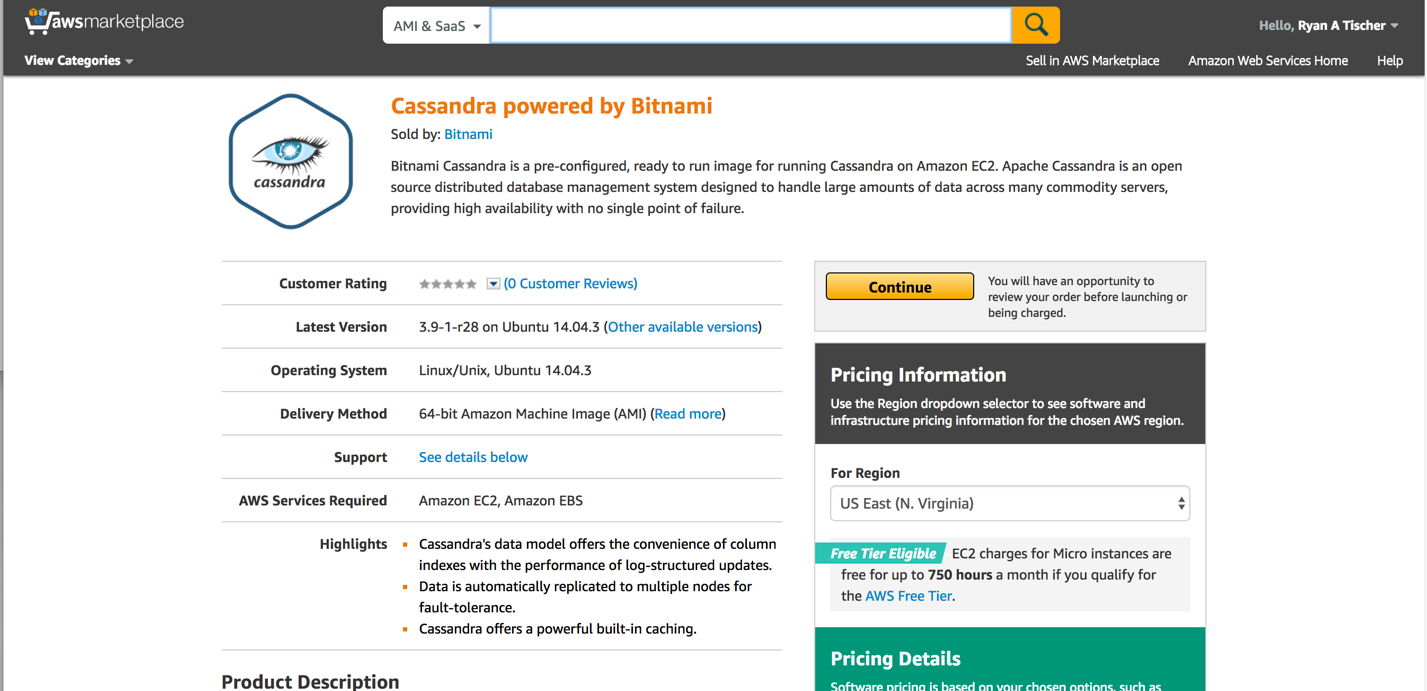
So following to be added as dependency in Zeppelin as shown below. Next step is to create notebook and run spark-cassandra data analysis explained in execution demo

com.datastax.spark:spark-cassandra-connector\_2.11:2.0.1

# 

# Infrastructure set up details(Cassandra):

Cassandra server is a pre-provisioned instance by Bitnami. The Bitnami image features Cassandra 3.9 running on Amazon machine image (Amazon Linux)



Connecting to the image requires a certificate file, in this case Tischer.pem. The certificate authenticates the session with the following command:

ssh -i "tischer.pem" [ubuntu@ec2-54-226-206-176.compute-1.amazonaws.com](mailto:ubuntu@ec2-54-226-206-176.compute-1.amazonaws.com)

Once connected access to Cassandra via the following command:

cqlsh -u cassandra -p KSns9DfaYahq

In this case the username is ‘cassandra’ and ‘KSns9DfaYahq’ is the password

**Cassandra configuration**

Cassandra is configured for a single keyspace called flightkeyspace

To following command creates the keyspace

CREATE KEYSPACE flightkeyspace WITH REPLICATION = { 'class':'SimpleStrategy', 'replication\_factor':1};

The following command uses the keyspace

USE flightkeyspace;

The following tables are created within the flightkeyspace

|  |  |
| --- | --- |
| **Table Name** | **Description** |
| flightiot | Holds flight IOT data, populated from Spark/kafka |
| customers | Holds customers on flights. Populated with Mock data |
| City\_data | Holds data about citys |
| Flightdelays | Holds customers + flight data with delays |

Tables are manually created using the following command. (Example for one table only.

create table city\_data (OriginAirportIDint,

DestAirportIDint,

destcityname text,

OriginCityName text,

origin text,

dest text,

PRIMARY KEY ((OriginAirportID, DestAirportID, destcityname,OriginCityName, origin, dest)));

Cassandra tables are further analyzed using zeppelin and Spark due to sheer volume cannot be easily queried using csqlh

In the event of moving data to or from CSV the following command was used:

copy flight\_test (OriginAirportID, DestAirportID, destcityname, OriginCityName origin, dest) to ‘temp12.csv’

Flight IOT table is populated from the streaming data. It is not meant for human consumstion. FlightIOT has the following columns.

uniquecarrier | flightnum | flightdate | origin | dest | actualelapsedtime | airlineid | airtime | arrdel15 | arrdelay | arrdelayminutes | arrivaldelaygroups | arrtime | arrtimeblk | cancellationcode | cancelled | carrier | carrierdelay | crsarrtime | crsdeptime | crselapsedtime | dayofmonth | dayofweek | departuredelaygroups | depdel15 | depdelay | depdelayminutes | deptime | deptimeblk | destairportid | destairportseqid | destcitymarketid | destcityname | deststate | deststatefips | deststatename | destwac | distance | distancegroup | diverted | firstdeptime | flights | lateaircraftdelay | longestaddgtime | month | nasdelay | originairportid | originairportseqid | origincitymarketid | origincityname | originstate | originstatefips | originstatename | originwac | quarter | securitydelay | tailnum | taxiin | taxiout | totaladdgtime | weatherdelay | wheelsoff | wheelson | year

**and example data**

AS | 657 | 2017-01-03 00:00:00.000000+0000 | SAT | SEA | 271 | 19930 | 251 | 0 | -16 | 0 | -2 | 2102 | 2100-2159 | 0 | 0 | AS | 0 | 2118 | 1840 | 278 | 3 | 2 | -1 | 0 | -9 | 0 | 1831 | 1800-1859 | 14747 | 1474703 | 30559 | Seattle, WA | WA | 53 | Washington | 93 | 1774 | 8 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 14683 | 1468303 | 33214 | San Antonio, TX | TX | 48 | Texas | 74 | 1 | 0 | N472AS | 6 | 14 | 0 | 0 | 1845 | 2056 | 2017

B6 | 1884 | 2017-01-26 00:00:00.000000+0000 | MCO | JFK | 140 | 20409 | 116 | 0 | -12 | 0 | -1 | 1517 | 1500-1559 | 0 | 0 | B6 | 0 | 1529 | 1301 | 148 | 26 | 4 | -1 | 0 | -4 | 0 | 1257 | 1300-1359 | 12478 | 1247803 | 31703 | New York, NY | NY | 36 | New York | 22 | 944 | 4 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 13204 | 1320402 | 31454 | Orlando, FL | FL | 12 | Florida | 33 | 1 | 0 | N907JB | 4 | 20 | 0 |

Customers table holds customer data is populated with Mock data from <https://www.mockaroo.com/>. Customers table has the following columns

flightnum | email | originairportid | destairportid | first\_name | id |last\_name | phone

and example data

flightnum | email | originairportid | destairportid | first\_name | id |last\_name | phone

-----------+-----------------------------+-----------------+---------------+------------+-----+-------------+-------------------

44 | ntibb22@prweb.com | 11042 | 11298 | Nissy| 75 | Tibb | 502-(644)575-3323

3042 | tdibbs71@123-reg.co.uk | 11603 | 12892 | Terese | 254 | Dibbs | 63-(350)531-3415

44 | rshepherdson15@cdc.gov | 11042 | 11298 | Rickie | 42 | Shepherdson | 7-(658)575-6810

3042 | owestmacott6p@dell.com | 11603 | 12892 | Orella | 242 | Westmacott | 81-(313)688-3710

4822 | jybarradp@toplist.cz | 12278 | 13487 | Janessa | 494 | Ybarra | 33-(422)984-2240

4674 | fhaughtonbe@soup.io | 13264 | 14869 | Fredrika | 411 | Haughton | 46-(477)414-8251

494 | cheyburn3@howstuffworks.com | 11697 | 11057 | Cally | 4 | Heyburn | 7-(434)411-8974

4822 | aforsythdc@webnode.com | 12278 | 13487 | Antonino | 481 | Forsyth | 86-(479)926-4568

44 | cwavell1o@dmoz.org | 11042 | 11298 | Cristionna| 61 | Wavell | 63-(765)701-1887

511 | awatting44@sfgate.com | 11298 | 11042 | Antin | 149 | Watting| 86-(764)715-5868

City Data holds City names and airport codes. It is populated from BTS data. City\_data table has the following columns

originairportid | destairportid | destcityname | origincityname | origin | dest

and example data

originairportid | destairportid | destcityname | origincityname | origin | dest

-----------------+---------------+------------------+--------------------+--------+------

14683 | 13487 | Minneapolis, MN | San Antonio, TX | SAT | MSP

10397 | 14633 | Rochester, MN | Atlanta, GA | ATL | RST

11618 | 13487 | Minneapolis, MN | Newark, NJ | EWR | MSP

12173 | 14831 | San Jose, CA | Honolulu, HI | HNL | SJC

13198 | 12339 | Indianapolis, IN | Las Vegas, NV | MCI | IND

14869 | 14679 | San Diego, CA | Salt Lake City, UT | SLC | SAN

12892 | 12339 | Indianapolis, IN | Los Angeles, CA | LAX | IND

14193 | 11042 | Nashville, TN | Pensacola, FL | PNS | CLE

10397 | 13485 | Madison, WI | Atlanta, GA | ATL | MSN

14122 | 12266 | Houston, TX | Pittsburgh, PA | PIT | IAH